Manual for Chameleon Basic 0.1

Based on the XGS Basic Manual

Updates for the Chameleon by David Betz

This manual describes the Chameleon Basic system, version 0.1.

Table of Contents

Typography Used in This Manual

The body of this manual is set using 12pt. Lucida Sans (a variable-width font).

Section headings are numbered and in bold face, and the text (but not the numbers) is underlined.

Subheadings, for "Syntax", "Examples", and similar topics are merely underlined.

Specifications of syntax, shown using BNF, are indented, set using Lucida Sans Typewriter (a fixed-width font) bold, for example:

$\textbf{\textit{expr}}_1$ **MOD** $\textbf{\textit{expr}}_2$

Within a syntax specification:

- Keywords are shown like **THIS**.
- Metavariables are shown like ***this***.[1]
- Metasyntactic characters (brackets, braces, vertical bars) are shown like `this`. (That is, monospace, but not bold and not italic.)

Within the body of text:

- A keyword is shown like **this**.
- A metavariable is shown like ***this***.
- A file name is shown like **THIS**.
- A product name is shown like **this**.

Examples of operators, statements, and functions are shown in 11pt. Lucida Sans Typewriter, for example:

```
MID$("abcdefghij", 3, 4)     "cdef"
```

Headings for such examples are shown in Lucida Sans, but *in italic and underlined*.

1 Note also that if there is more than one of a particular kind of metavariable, they are subscripted for reference in the description.

1 Introduction

This system consists of the following parts:

- the compiler (`CHAMBASIC.EXE`)
- the virtual machine

The compiler program is called `CHAMBASIC.EXE`. It is an MS-Windows console application; that is, in runs on MS-DOS, or under MS-Windows but only using the COMMAND or CMD application. You will need to use one of those methods to navigate to the folder and run command lines arguments. A full description of the command line syntax is given below.

The virtual machine (VM) runs on either the Chameleon PIC or the Chameleon AVR. To run Chameleon Basic programs, you must first flash the Chameleon Basic virtual machine into your Chameleon PIC or Chameleon AVR board. See sections 1.1 and 1.2 for details.

1.1 Flashing the Virtual Machine

To use Chameleon Basic you must first flash the Chameleon Basic virtual machine into your Chameleon board.

1.1.1 for the Chameleon PIC

1. Open a "Command Prompt" window
2. Change directory to the chambasic folder created by unzipping chambasic.zip
3. Connect your Chameleon PIC board to your PC using a USB cable
4. Determine the port number of the COM port associated with the Chameleon PIC board
5. Invoke the "flashpic.bat" batch file passing it the COM port number as an argument. For example, if the Chameleon PIC board is on COM10, type "flashpic 10".

1.1.2 for the Chameleon AVR

6. Open a "Command Prompt" window
7. Change directory to the chambasic folder created by unzipping chambasic.zip
8. Connect your Chameleon AVR board to your PC using a USB cable
9. Determine the port number of the COM port associated with the Chameleon AVR board
10. Invoke the "flashavr.bat" batch file passing it the COM port number as an argument. For example, if the Chameleon AVR board is on COM10, type "flashavr 10".

2 Command Line Syntax

Use the following syntax to compile an Chameleon Basic source file:

```
     CHAMBASIC [ -b ca|cp ] -p port ] [ -t ] input-file
```

Use the **-p** option to specify the serial port to use to download the compiled program to the PIC or AVR. If you don't provide this option, the default port is "COM4". The port will be configured to 9600 baud, 8 bits no parity. To select “COM2”, enter either -**pCOM2** or -**p2**.

- Use **-bca** to compile for the Chameleon AVR board.
- Use **-bcp** to compile for the Chameleon PIC board.
- Use **-c** to compile a file without downloading it.
- Use **-t** to enter terminal mode after the download completes.

The ***input-file*** parameter is required as is the **-b** `to select the target board`.

The compiler produces an output file with the same name as the input file but with the extension **.BAI** in place of **.BAS**.

2.1 The Game of Life demo[2]

To load this sample program, connect your XGS PIC to your PC using the USB2Serial adapter and determine the COM port that gets assigned to the adapter. Insert a formatted microSD card into the XGS PIC and type the following command:

```
XGSBASIC -p com2 LIFE.BAS
```

This should compile the `LIFE.BAS` demo program and download it to the microcontroller's memory. Because no `-c`, `-f`, or `-r` option is provided, `-f` is assumed, and the program is downoaded to the internal flash.

After that, just reset the XGS PIC and the program should start. A random pattern will be generated and then the "life" algorithm will be run for 30 generations. After that, a new random pattern will be generated and the process starts over. If you like what is unfolding with a particular pattern, you can press the yellow button on the gamepad and that will extend that pattern beyond the normal 30 generations. Just keep holding the button down until you're tired of watching that pattern. When you release the button, a new pattern will be generated.

2 Note that the LIFE.BAS program is specific to, and will run only on, the XGS PIC.

3 Language Syntax

Names:

Before going further, a discussion of *names* is in order. Several kinds of things are identified by names: *variables* (both scalars and arrays); *constants*; statement *labels*; and *subroutines* and *functions*.

A name can be at most 32 characters long. A name must start with a letter but can contain letters of either case, digits, "$", "%" and "_". Also, a variable name that ends with "$" is assumed to be a string variable unless otherwise specificed; similarly, a variable name that ends with "%" is assumed to be an integer variable.

It should be noted that Chameleon Basic is not case-sensitive: the names “foo”, “Foo”, and “FOO” are all the same name as far as it is concerned.

Programs in Chameleon Basic consist of statements. Each statement occupies a single line, and each line consists of a single statement.

Expressions:

Expressions are used in many of the statements of this language. While there are some statements that are so simple that they do not require any expressions, expressions are so fundamental that we will discuss them before discussing the actual statements.

Statements:

Any statement may be preceded by a *label*. Doing so is required for some purposes, but most lines do not require them, and should not have them. This dialect of BASIC does not support the concept of *line numbers*. The use of labels will be discussed later, as necessary.

Some statements are not complete in and of themselves, and must be used in groups, or at least in pairs. For example, the **`SUB`** statement begins the definition of a subroutine.
The **`END SUB`** statement is ends the definition. All statements in between the two statements are a part of that subroutine.

3.1 Expressions

An expression is either a constant, a variable, or some combination of one or more of those using various operators. There are so many that it is helpful to consider them in groups or categories:

- Constant Expressions
  decimal-constant **0x*hex-constant*** ***string-constant***

- Arithmetic Expressions
  + - * / MOD -

- Logical Expressions
  NOT OR AND = <> < <= >= >

- Bitwise Expressions
  ~ & | ^ << >>

- Other Expressions
  (...) ***variable*** ***array-reference*** ***function-call***

Below are descriptions of each of them, by category. We examine constant expressions first, because we will see them in examples of all the other expressions.

3.1.1 Constant Expressions

decimal-constant ***hexadecimal-constant*** ***string-constant***

3.1.1.1 decimal constant

The value of this expression is a specific integer value represented as a signed decimal number.

Syntax:

`[ { + | - } ]` ***decimal-digit-string***

where ***decimal-digit-string*** is from 1 to 5 decimal digits with no intervening characters of any kind. The lower bound is -32768, and the upper bound is 32767.

A "minus" (negative-value) symbol may precede the ***decimal-digit-string***, and space is allowed between the sign and the ***decimal-digit-string***.

Examples:

This expression:	has this value:
`0`	`0`
`-0`	`0`
`000`	`0`
`9`	`9`
`09`	`9`
`-9`	`-9`
`-09`	`-9`
`32767`	`32767`
`-  32768`	`-32768`

Counterexamples:

`32768` *(too large a positive value)*	`-32768`
`-32769` *(too large a negative value)*	`32767`

3.1.1.2 hexadecimal constant

The value of this expression is a specific integer value represented as a unsigned hexadecimal number.

Syntax:

```
    [ { + | - } ] 0xhex-digit-string
```

where ***hex-digit-string*** is from 1 to 4 hexadecimal digits with no intervening characters of any kind. The lower bound is `0000`, and the upper bound is **`FFFF`**.

The two-character string **`0x`** prefixes the hexadecimal number in order to let the compiler (and a subsequent human reader) know that any decimal digits are actually part of a base-sixteen number. No space is allowed between the two characters `0` and **`x`** or between the **`x`** and the ***hex-digit-string***. The letter **`x`** must be in lower case; it must not be a capital or upper-case **`X`**.

A "minus" (negative-value) symbol may precede the **`0x`**, and space is allowed between the sign and the **`0x`**.

Examples:

This expression:	has this value:
0x0	0
0x00000	0
0x9	9
0x00009	9
0xF	15
0x0000F	15
0xFF	255
0xFFF	4095
0x7FFF	32767
0x8000	-32768
0xFFFF	-1
0xFFFFF	-1
0xf	15
0x0000f	5
-0xf	-15
0Xf *(upper-case **X**)*	*(none)*
0xG *(invalid hex digit)*	*(none)*

3.1.1.3 string constant

The value of this expression is a specific sequence of printable characters.

Syntax:

" ***printable-characters*** "

The printable characters include the blank (0x20) as well as punctuation, digits, and upper-case and lower-case letters.

The characters of the string must be enclosed in a pair of double-quotes (”...”); a double-quote character may be included in the string by preceding it with a backslash (in the manner of the C language and its descendents).

Examples:

This statement:	produces this output:
PRINT "ABcd 09 ,.;:!?"	ABcd 09 ,.;:!?
PRINT "'ABcd 09 ,.;:!?'"	'ABcd 09 ,.;:!?'
PRINT "\"ABcd 09 ,.;:!?\""	"ABcd 09 ,.;:!?"

3.1.2 Arithmetic Expressions

Arithmetic expressions include those with the following operators:

```
+  -  *  /  MOD  -
```

3.1.2.1 addition[3]

Syntax:

$expr_1$* + *$expr_2$

This expression adds ***$expr_2$*** to ***$expr_1$***. number.

Example:

The value of the following expression is 11:

6 + 5

3.1.2.2 subtraction

Syntax:

$expr_1$* – *$expr_2$

This expression subtracts ***$expr_2$*** from ***$expr_1$***.

Example:

The value of the following expression is 6:

11 - 5

3.1.2.3 multiplication

Syntax:

expr1 * expr2

This expression multiplies ***$expr_2$*** by ***$expr_1$***.

Example:

The value of the following expression is 30:

6 * 5

3.1.2.4 division

Syntax:

$expr_1$* / *$expr_2$

This expression divides ***$expr_1$*** by ***$expr_2$***.

Example:

The value of the following expression is 6:

30 / 5

3 The **+** operator is also used as the string concatenation operator. Note that it cannot be used to add or concatenate a string and a number. It can only be used to add two numeric expressions or to concatenate two strings.

3.1.2.5 modulo

Syntax:

```
expr1 MOD expr2
```

This expression divides `expr1` by `expr2`, and returns the remainder.

Example:

The value of the following expression is 0:

```
30 MOD 5
```

The value of the following expression is also 0:

```
30 MOD 6
```

The value of the following expression is 1:

```
31 MOD 5
```

The value of the following expression is 2:

```
30 MOD 4
```

3.1.2.6 negation

Syntax:

```
- expr
```

This expression negates, or returns the negative value of, ***expr***.

Example:

The value of the following expression is negative three:

```
- 3
```

3.1.3 Relational Expressions

= <> < <= > >=

Relational expressions make arithmetic comparisons between numbers. They return 0 (zero) to represent FALSE and 1 (one) to represent TRUE.

3.1.3.1 equality

The value of this expression is 1 (one) if the specified expressions are equal to each other; otherwise, the value is 0 (zero).

Note: *This expression should not be confused with the assignment statement!*

Syntax:

$\textbf{\textit{expr}}_1$ **=** $\textbf{\textit{expr}}_2$

Examples:

The value of the following expression is 0 (representing FALSE):

```
3 = 2
```

The value of the following expression is 1 (representing TRUE):

```
4 = 4
```

3.1.3.2 inequality

The value of this expression is 1 (one) if the specified expressions are *not* equal to each other; otherwise, the value is 0 (zero).

Syntax:

$\textbf{\textit{expr}}_1$ **<>** $\textbf{\textit{expr}}_2$

Examples:

The value of the following expression is 0 (representing FALSE):

```
3 <> 3
```

The value of the following expression is 1 (representing TRUE):

```
3 <> 4
```

3.1.3.3 less-than

The value of this expression is the 1 (one) if the value of ***expr***$_1$ is strictly less than the value of ***expr***$_2$; otherwise, the value is 0 (zero).

Syntax:

expr$_1$ **<** ***expr***$_2$

Examples:

The value of the following expression is 0 (representing FALSE):

4 < 3

The value of the following expression is 0 (representing FALSE):

4 < 4

The value of the following expression is 1 (representing TRUE):

4 < 5

3.1.3.4 less-than-or-equal-to

The value of this expression is the 1 (one) if the value of ***expr***$_1$ is less than *or equal to* the value of ***expr***$_2$; otherwise, the value is 0 (zero).

Syntax:

expr$_1$ **<=** ***expr***$_2$

Examples:

The value of the following expression is 0 (representing FALSE):

4 <= 3

The value of the following expression is 1 (representing TRUE):

4 <= 4

The value of the following expression is 1 (representing TRUE):

4 <= 5

3.1.3.5 <u>greater-than</u>

The value of this expression is the 1 (one) if the value of ***expr***$_1$ is strictly greater than the value of ***expr***$_2$; otherwise, the value is 0 (zero).

<u>Syntax:</u>

expr$_1$ **>** ***expr***$_2$

<u>Examples:</u>

The value of the following expression is 0 (representing FALSE):

6 > 7

The value of the following expression is 0 (representing FALSE):

7 > 7

The value of the following expression is 1 (representing TRUE):

7 > 6

3.1.3.6 <u>greater-than-or-equal-to</u>

The value of this expression is the 1 (one) if the value of ***expr***$_1$ is greater than *or equal to* the value of **expr**$_2$; otherwise, the value is 0 (zero).

<u>Syntax:</u>

expr$_1$ **>=** ***expr***$_2$

<u>Examples:</u>

The value of the following expression is 0 (representing FALSE):

6 >= 7

The value of the following expression is 1 (representing TRUE):

7 >= 7

The value of the following expression is 1 (representing TRUE):

7 >= 6

3.1.4 Logical Expressions

```
NOT  OR  AND
```

Logical expressions treat 0 (zero) as FALSE and *any non-zero value* as TRUE. Similarly, they return 0 (zero) to represent FALSE and 1 (one) to represent TRUE.

Note: *These are not the same as bitwise operations with the same or similar names.* Logical operators perform their operations on the whole value of each expression, and return either an integer 0 (zero) or an integer 1 (one); bitwise operators (see below) perform their operations on corresponding bits in each of the expressions, and return a new integer representing those result of those operations.

3.1.4.1 logical NOT

The value of this expression is TRUE if the specified expression is FALSE , and is FALSE if the specified expression is TRUE.

Syntax:

```
NOT expr
```

Examples:

The value of the following expression is 1 (representing TRUE):

NOT 0

The value of the following expression is 0 (representing FALSE):

NOT 3

3.1.4.2 logical OR

The value of this expression is TRUE if the values of either (or both) of the specified expressions is (or are) TRUE.

Note: This operator uses "short-circuit evaluation". That is, if ***expr***$_1$ is **`TRUE`**, then ***expr***$_2$ is never even evaluated, and the entire expression evaluates to TRUE .

Syntax:

```
expr₁ OR expr₂
```

Examples:

The value of the following expression is 0 (representing FALSE):

```
0 OR 0
```

The value of the following expression is 1 (representing TRUE):

```
0 OR 3
```

The value of the following expression is 1 (representing TRUE):

```
-12 OR 0
```

The value of the following expression is 1 (representing TRUE):

```
-11 OR 1
```

3.1.4.3 logical AND

The value of this expression is TRUE if the values of both of the specified expressions are TRUE.

Note: This operator uses "short-circuit evaluation". That is, if ***expr***$_1$ is **`FALSE`**, then ***expr***$_2$ is never even evaluated, and the entire expression evaluates to FALSE.

Syntax:

```
expr₁ AND expr₂
```

Examples:

The value of the following expression is 0 (representing FALSE):

```
0 AND 0
```

The value of the following expression is 0 (representing FALSE):

```
0 AND 3
```

The value of the following expression is 0 (representing FALSE):

```
-12 AND 0
```

The value of the following expression is 1 (representing TRUE):

```
-11 AND 1
```

3.1.5 Bitwise Expressions

~ & | ^ << >>

3.1.5.1 bitwise NOT

The value of this expression is the integer representation of the inversion, or ones-complement, of the bits of the specified expression.

Syntax:

```
~ expr
```

Examples:

This expression:	has this value:
~ 0	-1
~ 0x0000	0xFFFF
~ -1	0
~ 0xFFFF	0x0000
~ -2	1
~ 0xFFFE	0x0001
~ -256	255
~ 0xFF00	0x00FF
~ -275	274
~ 0xFEED	0x0112

3.1.5.2 bitwise inclusive OR

The value of this expression is the integer representation of the inclusive OR of the corresponding bits of the specified expressions. That is, if a given bit in ***expr₁*** is set to 1 *or* the corresponding bit in ***expr₂*** is set to 1, or *both bits are set*, then the corresponding bit in the result is set to 1; otherwise, it is set to 0 (zero).

Syntax:

expr₁ | ***expr₂***

Examples:

This expression:	has this value:
0 | 1	1
0x0000 | 0x0001	0x1
1 | 2	3
0x0001 | 0x0002	0x0003
2 | 3	3
0x0002 | 0x0003	0x0003
-256 | 255	-1
0xFF00 | 0x00FF	0xFFFF

3.1.5.3 bitwise exclusive OR

The value of this expression is the integer representation of the exclusive OR of the corresponding bits of the specified expressions. That is, if a given bit in ***expr₁*** is set to 1 *or* the corresponding bit in ***expr₂*** is set to 1, *but not both bits are set*, then the corresponding bit in the result is set to 1; otherwise, it is set to 0 (zero).

Syntax:

```
expr1 ^ expr2
```

Examples:

This expression:	has this value:
`0 ^ 1`	`1`
`0x0000 ^ 0x0001`	`0x0001`
`1 ^ 2`	`3`
`0x0001 ^ 0x0002`	`0x0003`
`2 ^ 3`	`1`
`0x0002 ^ 0x0003`	`0x0001`
`65280 ^ 255`	`-1`
`0xFF00 ^ 0x00FF`	`0xFFFF`
`43690 ^ 21845`	`-1`
`0xAAAA ^ 0x5555`	`0xFFFF`
`43690 ^ 65280`	`21930`
`0xAAAA ^ 0xFF00`	`0x55AA`

3.1.5.4 bitwise AND

The value of this expression is the integer representation of the AND of the corresponding bits of the specified expressions. That is, if a given bit in ***expr**₁* is set to 1 *and* the corresponding bit in ***expr**₂* is set to 1, then the corresponding bit in the result is set to 1; otherwise, it is set to 0 (zero).

Syntax:

expr₁* & *expr₂

Examples:

This expression:	has this value:
`0 & 1`	`0`
`0x0000 & 0x0001`	`0x0000`
`1 & 2`	`0`
`0x0001 & 0x0002`	`0x0000`
`2 & 3`	`2`
`0x0002 & 0x0003`	`0x0002`
`-256 & 255`	`0`
`0xFF00 & 0x00FF`	`0x0000`
`-21846 & 21845`	`0`
`0xAAAA & 0x5555`	`0x0000`
`-21846 & -256`	`-22016`
`0xAAAA & 0xFF00`	`0xAA00`

3.1.5.5 bitwise shift left

The value of this expression is the integer representation of shifting ***expr**₁* left by the number of bits specified by ***expr**₂*.

Syntax:

```
expr1 << expr2
```

Examples:

This expression:	has this value:
`1 << 1`	`2`
`0x0001 << 0x0001`	`0x0002`
`1 << 2`	`4`
`0x0001 << 0x0002`	`0x0004`
`1 << 8`	`256`
`0x0001 << 0x0008`	`0x0100`
`15 << 4`	`240`
`0x000F << 0x0004`	`0x00F0`
`15 << 8`	`3840`
`0x000F << 0x0008`	`0x0F00`
`255 << 8`	`-256`
`0x00FF << 0x0008`	`0xFF00`
`255 << 16`	`0`
`0x00FF << 0x0010`	`0x0000`

3.1.5.6 bitwise shift right

The value of this expression is the integer representation of shifting ***$expr_1$*** right by the number of bits specified by ***$expr_2$***.

Note: This is an *arithmetic* shift. Hence, the sign bit (the most-significant bit) is preserved, and is also copied to the next bit to its right, for as many bits as specified by ***$expr_2$***.

Syntax:

$expr_1$* >> *$expr_2$

Examples:

This expression:	has this value:
`1 >> 1`	`0`
`0x0001 >> 0x0001`	`0x0000`
`1 >> 2`	`0`
`0x0001 >> 0x0002`	`0x0000`
`2 >> 1`	`1`
`0x0002 >> 0x0001`	`0x0001`
`15 >> 1`	`7`
`0x000F >> 0x0001`	`0x0007`
`240 >> 4`	`15`
`0x00F0 >> 0x0004`	`0x000F`
`-256 >> 8`	`-1`
`0xFF00 >> 0x0008`	`0xFFFF`
`-256 >> 16`	`-1`
`0xFF00 >> 0x0010`	`0xFFFF`

3.1.6 Other Expressions

(...) ***variable*** ***array-reference*** ***function-call***

3.1.6.1 parentheses

The value of this expression is the value of the expression inside the matched pair of parentheses.

Syntax:

(*expr*)

Parentheses simply provide the traditional way of grouping expressions together, particularly for the purpose of over-riding operator precedence.

Examples:

This expression:	has this value:
`6 / 2 + 4`	7
`( 6 / 2 ) + 4`	7
`6 / (2 + 4)`	1
`4 + 6 / 2`	7
`( 4 + 6 ) / 2`	5

3.1.6.2 variable

A variable is simply a value that changes, while the variable *name* remains the same.

Syntax:

variable

Examples:

This expression or statement:	has this value or does this:
x	(undefined!)
LET x = 6	assigns x the value 6
LET y = 2	assigns y the value 2
z = 4	assigns z the value 4
x	6
y	2
z	4
x / y	3
y + z	6
x / y + z	7
(x / y) + z	7
x / (y + z)	1
z + x / y	7
(z + x) / y	5

3.1.6.3 array reference or element

An *array* is simply a variable that can contain or represent more than one value simultaneously, each one distinguished from the others by its index (or subscript). The index may be any expression whose value is an integer; that is, it may not be a floating-point value or a string.

Note: In keeping with other dialects of BASIC, and by contrast with C, array indexes in Chameleon Basic start with 1 (one), not 0 (zero).

Generally, an array is used to group together two or more values that are in some sense alike, for instance, the highest temperature on each day of the year, or the wave frequency of each note in a scale or tune.

Syntax:

```
    variable ( index [ , index ] )
```

Example:

Suppose your program includes the following statements:

```
    LET piano(40) = 261                // C4
    LET piano(41) = 277                // C#4 or Db4
    LET piano(42) = 293                // D4
    LET piano(43) = 311                // D#4 or Eb4
    LET piano(44) = 329                // E4
    LET piano(45) = 349                // F4
    LET piano(46) = 369                // F#4 or Gb4
    LET piano(47) = 391                // G4
    LET piano(48) = 415                // G#4 or Ab4
    LET piano(49) = 440                // A4
    LET piano(50) = 466                // A#4 or Bb4
    LET piano(51) = 493                // B4
    LET piano(52) = 523                // C5
```

This stores the frequencies of the musical pitches noted in the comments into a set of array elements. (Yes, those frequencies are approximate.) The index of each array element is the piano key corresponding to that pitch.

You might then define a two-dimensional array to contain a sequence of notes. Each element of this array would have two parts – the frequency, and the duration. Something like this:

```
    tada(1,1) = piano(52)              // “TA” on High C ...
    tada(1,2) = 483                    // for almost half a second.
    tada(2,2) = 0                      // Silence ...
    tada(2,2) = 17                     // for just a jiffy.
    tada(3,1) = piano(52)              // “DA” on High C ...
    tada(3,2) = 1500                   // for 1.5 seconds.
```

3.1.6.4 function call

The value of a function call is the value of the name of the function immediately prior to ending (or returning, or exiting). See the section later in this document regarding how to define a function..

Syntax:

```
name ( [ arg [ , arg ] ... ] )
```

or

```
name
```

That is, the parentheses are optional *if there are no arguments*.The **name** is just the name of the function.

There can be any number of arguments, even none at all, as long as they match they number of arguments with which the function was defined.

Each argument can be any expression, as long as it matches the type of expression of the corresponding argument with which the function was defined.

Example:

Suppose your program contains the following statements, which define a function that computes the area of a right triangle, given the two orthogonal sides.

```
DEF rightTriangleArea ( side1, side2 )
rightTriangleArea = side1 * side2 / 2
END DEF
```

This function could then be called as follows:

```
LET A =  rightTriangleArea ( 3, 4 )
```

which would set the variable “A” to the value 3*4/2, or 6. Or it could be called this way:

```
PRINT  rightTriangleArea(9,8)
```

which would display the number 36 (that is, 9*8/2) on a line by itself.

Now we are ready to consider the statements that use all these expressions.

3.2 Simple Statements

Here is a list of statements that stand by themselves:

```
REM
OPTION
DEF[4]
DIM
IF[5]
LET
GOTO
CALL
PRINT
STOP
END
```

Here are descriptions of each of them:

3.2.1 REM

Syntax:

```
REM [ comment text to end of line ]
```

This statement is used to include remarks or comments in the program. They are completely ignored by the compiler, and do not show up in compiled (and downloaded and executed) program in any form. They are included in a program as a means of communicating to some other programmer (or oneself!) at a future time what a certain part of the program is supposed to do, or what algorithm is being used, or something of that sort.

Comments can also be included using the syntaxes common to C and many other languages:

```
// [ comment text to end of line ]
/* [ comment text ]
        [ between slash+asterisk pair ]
        [ and matching asterisk+slash pair ]
*/
```

4 There are two forms of the **DEF** statement. One is a simple statement, requiring no other statements to be complete. That form is described in this section. The other form requires a matching **END DEF** statement, and is described in the **Compound Statements** section, below.

5 There are two forms of the **IF** statement. One is a simple statement, requiring no other statements to be complete. That form is described in this section. The other form requires a matching **END IF** statement, and may also include **ELSE** or **ELSE IF** statements, and is described in the **Compound Statements** section, below.

Examples:

```
REM The following takes place
REM on the day of the Massachusetts primary election.
REM It is the shortest day of my career.
```

or

```
/*
The following takes place
on the day of the Massachusetts primary election.
It is the shortest day of my career.
*/
```

or

```
LET a=  3              // Set variable to length of one side.
LET b=  4              // Set variable to length of other side.
LET csq=(a*a) + (b*b)  // Set variable to sum of squares of sides.
```

3.2.2 OPTION

This statement is the way to set various compiler options.

Note: This statement must be the first statement of the program.

Syntax:

```
OPTION option [ (...) ]
```

Options are:

```
TARGET
HEAP
```

Descriptions follow.

3.2.2.1 TARGET

This option selects either the "chameleon" or "xgs" runtime environments. The “xgs” target is not yet supported.

```
TARGET = { “chameleon” | “xgs” }
```

3.2.2.2 HEAP

This option sets the heap size to something other than the default.

Syntax:

```
HEAP "size"
```

or

```
HEAP "size:handles"
```

The ***size*** parameter is the amount of space in bytes to be reserved for storing strings. Each string takes up 6 bytes plus the number of characters in the string. The ***handles*** parameter is the total number of strings that can be in use at any one time. Any string that is not the empty string or a string literal takes one handle. This includes temporary strings created during the evaluation of expressions.

The default string heap size for the PIC is "1024:32". The default heap size for the AVR is "512:16". If you specify just the size, the number of handles defaults to the heap size divided by 32.

If you don't need dynamic string support at all, you can turn it off by setting the size to zero with the following statement:

```
OPTION HEAP "0"
```

3.2.3 DEF

Syntax:

```
    DEF name = value
```

This form of the **DEF** statement is self-contained, and merely defines a constant; that is, it defines a name to have an unchangeable value.

Example

The following defines “hundredpi” to be a constant whose value is always (roughly) 100 times the value of **π.**

```
    DEF hundredpi = 314
```

3.2.4 DIM

Syntax:

```
    DIM variable-defs
```

See section "variable-defs", below.

This statement is the way to declare one or more either scalar or array variables. The initializers may be spread over more than one line.

Examples:

```
    DIM A
    DIM A = 1
    DIM B(3)
    DIM B(3) = { 1, 2, 3 }
```

3.2.5 LET

Syntax:

```
    [ LET ] l-value = expr
```

This is the assignment statement. It assigns the expression to the right of the “equals” sign to the l-value on the left. An l-value is just a way of saying something that can have a value assigned to it, i.e. either a scalar (one-dimensional) variable or a single element of an array.

Note that the word **LET** is optional. However, if present, it must be the first word of the statement, and no other word may be there instead.

Example:

```
    LET A = 7
    pixels_per_brick = 47
    let ballWidth=15
```

3.2.6 IF

Syntax:

```
IF expr THEN statement
```

This statement is a way for a program to do a thing or not do a thing.

Examples:

If a value is zero, set it to some specific (default) value:

```
IF number_of_monsters = 0 THEN LET number_of_monsters = 111
```

Similarly, if some counter has reached a predetermined maximum, set it back to one.

```
IF N >= 24 THEN N = 1
```

3.2.7 GOTO

Syntax:

```
GOTO label
```

This statement causes the program execute the statement at "label" instead of executing the statement immediately following the **GOTO** statement. The **GOTO** statement seems obvious and innocent at first, but has generally been found to cause complexity and confusion if used more than sparingly. The Chameleon Basic language has many ways to organize sequences of statements in an orderly way, so the **GOTO** statement should be easy to avoid in most cases.

Note: **GOTO** in the main code can refer only to labels in the main code. **GOTO** within a function or subroutine can refer only to labels within the same function or subroutine.

Example:

```
LET x=1
abc:  LET x=x+1
GOTO hijk
efg:  LET x=x-5
GOTO efg
hijk: LET x=x+2
STOP
END
```

Two questions immediately arise: (1) Does this program ever finish? (2) What is the value of **x** if and when it does?

3.2.8 CALL

This statement calls the specified subroutine with the specified arguments (if any).

Syntax:

```
[ CALL ] name ( [ arg [ , arg ] ... ] )
[ CALL ] name
```

That is: (a) the `CALL` keyword is optional, and (b) the parentheses are optional *if there are no arguments*.

If there is more than one argument, each argument must be separated from the next by using a comma. Note that the ellipsis (three periods in a row) just means that there may be an arbitrary number of arguments, each (except the first) preceded by a comma. See subsection **SUB**, under **Compound Statements**, below.

Example:

```
SUB sayhi
PRINT "Hello, World!"
END SUB
CALL sayhi ()
CALL sayhi
sayhi
```

This example defines a subroutine (see below) called "sayhi", which merely displays the text string "Hello, World!". Following the definition, the subroutine is *called* three times in a row; each time, it displays the same message.

3.2.9 PRINT

This statement sends text to the serial interface. To send text to the screen, see the `DISPLAY` statement, below.

Syntax:

```
PRINT [ expr [ [ { , | ; } expr ] (...) ] ]
```

The text will represent zero or more expressions, as specified in the statement. Each expression may be a string or decimal or hexadecimal constant, or a scalar variable, or an array element. If no expressions are included, a blank line is displayed. If only one expression is included, no other syntax is required. If more than one expression is included, each must be separated from the next by either a comma or a semicolon.

If the separator is a semicolon, the second expression will appear immediately adjacent to the previous expression; in effect, they will *appear* to be concatenated.

On the other hand, if the separator is a comma, the second expression will begin at the next 8th column on the line.

Examples:

Print an empty or blank line:

```
PRINT
```

Print the number “7” on a line by itself:

```
LET A = 7
PRINT A
```

Print “4715” on a line by itself:

```
LET pixels_per_brick = 47
LET ballWidth = 15
PRINT pixels_per_brick ;  ballWidth
```

Print “47   15” (that is “47” followed by 6 blanks or spaces, followed by “15”) on a line:

```
PRINT pixels_per_brick ,  ballWidth
```

3.2.10 DISPLAY

This statement sends text to the screen. To send text to the serial interface, see the **PRINT** statement, above.

Syntax:

DISPLAY [***expr*** [[{ , | ; } ***expr***] (...)]]

This statement has the same syntax and semantics as the PRINT statement.

3.2.11 STOP

Syntax:

STOP

This statement tells the program to stop altogether, regardless of where in the program it appears or how it was encountered.

3.2.12 END

Syntax:

END

This statement tells the compiler that it is the last statement of the program. It has no effect on the program at run time. It is optional, but its use is encouraged.

3.3 Compound Statements

Here is a list of statements that must appear in groups:

DEF[6]
END DEF
SUB
END SUB
IF[7]
ELSE IF
ELSE
END IF
SELECT
CASE
CASE ELSE
END SELECT
FOR
NEXT
DO
LOOP

Here are descriptions of each of them:

6 There are two forms of the **DEF** statement. One is a simple statement, requiring no other statements to be complete. That form is described in the **Simple Statements** section, above. The other form requires a matching **END DEF** statement, and is described in the this section.

7 There are two forms of the **IF** statement. One is a simple statement, requiring no other statements to be complete. That form is described in the **Simple Statements** section, above. The other form requires a matching **END DEF** statement, and may also include **ELSE** or **ELSE IF** statements, and is described in the this section.

3.3.1 DEF

This form of the **DEF** statement defines a function.

Syntax:

```
DEF name [ ( [ arg [ , arg ] ... ] ) ]
...
END DEF
```

The statement itself (with the name and parentheses and arguments) specifies how the function will be called. It must be followed by a matching **END DEF** statement (as shown). All the statements in between specify what the function does to achieve the result that it returns. In this form, the **END DEF** statement is *required*.

Note that if the function does not use any arguments, the entire argument list including the parentheses may be omitted.

Inside of the function, the function's name is used as a variable to which to assign the return value; the value of that variable at the time the function completes execution is the return value of the function. There is no RETURN statement, as in some other dialects of BASIC.

Examples:

The following defines a function that computes the area of a right triangle, given the two orthogonal sides. The “body” of the function consists of just two statements, which compute the area of the square and divides that by 2, and assigns that the name of the function.

```
DEF rightTriangleArea ( side1, side2 )
rightTriangleArea = side1 * side2
rightTriangleArea = rightTriangleArea / 2
END DEF
```

The body of this function could just as easily be written as a single line, as follows:

```
DEF rightTriangleArea ( side1, side2 )
rightTriangleArea = side1 * side2 / 2
END DEF
```

This function could then be called as follows:

```
LET A =  rightTriangleArea ( 3, 4 )
```

which would set the variable “A” to the value 6. Or it could be called this way:

```
PRINT  rightTriangleArea(9,8)
```

which would display the number 36 on a line by itself.

3.3.2 <u>SUB</u>

This statement defines a subroutine.

<u>Syntax:</u>

```
SUB name ( [ arg [ , arg ] ... ] )
END SUB
```

A subroutine is essentially the same as a function, except that it does not return a value (or "have a return value"). See subsection CALL, under **Simple Statements**, above.

<u>Example:</u>

The following defines a subroutine that chooses a "fortune" at random, and displays it. Note that it does not return the value of the fortune for any further processing. (Refer to the example in section "array reference or element", and assume that the array "piano" has been set the same way.)

```
SUB playRandomNote ()
freq = piano[ 40 + RND(13)-1 ]
SOUND( freq )
END DEF
```

This subroutine could be called as follows:

```
CALL  playRandomNote ()
```

or:

```
CALL  playRandomNote
```

or even just:

```
playRandomNote
```

This would have the effect of playing a random note on the scale continaing middle C.

3.3.3 IF

Syntax:

```
IF expr THEN statement
IF expr THEN
  [ ELSE IF expr THEN ]
  [ ELSE ]
  END IF
```

This statement is the way for a program to do different things instead of each other, depending on circumstances.

The simplest case provides the means to either do a thing or not do a thing. The second form provides a way to do several things, or not do them; or to do more than one alternative thing or set of things.

Examples:

If a value is zero, set it to some specific (default) value:

```
IF number_of_monsters = 0 THEN LET number_of_monsters = 111
```

Similarly, if some counter has reached a predetermined maximum, set it back to one.

```
IF N >= 24 THEN N = 1
```

If you need to do more than one thing (or not), use this form:

```
IF number_of_monsters = 0 THEN
  LET level = level + 1
  LET number_of_monsters = 111 * level
END IF
```

If you need to do two different things depending on circumstances, use this form:

```
DEF furry = 1
DEF flying = 2
IF level MOD 2 = 1 THEN
  monster_type = furry
ELSE
  monster_type = flying
END IF
```

If you need to do more than two different things, the IF ... THEN ... ELSE IF chain may be your answer:

```
DEF Sunday = 1
DEF Monday = 2
...
DEF Saturday = 7
IF (dayOFweek = Saturday)
  PRINT "Have a nice weekend!"
ELSE IF (dayOFweek = Sunday)
  PRINT "Have a nice Sunday!"
ELSE
  PRINT "Have a nice day!"
```

(This example is based on one in the PHP sections of the w3schools.com web site.)

One IF statement can be “nested” inside another:

```
DEF furry = 1
DEF flying = 2
DEF slimy = 3
DEF arach = 4
IF level MOD 2 = 1 THEN
  IF LEVEL > 5 THEN
     monster_type = furry
  ELSE
     monster_type = slimy
  END IF
ELSE
  IF level > 5
     monster_type = arach
  ELSE
     monster_type = flying
  END IF
END IF
```

3.3.4 SELECT

Syntax:

```
SELECT expr0
  [ CASE case-expr [ , case-expr ] (...)
        statements
        ]
  (...)
  [ CASE ELSE
        statements
        ]
END SELECT
```

This statement performs one or more different statements (or sequences of statements) based on whether ***expr***$_0$ matches any of the values in the **CASE** statements.

Each **CASE** statement (except the **ELSE** variant) includes one (or more) *case expressions*. If there are more than one, each is separated from the one before it by a comma. Each ***case-expr*** can be either an individual value or a range of value, i.e.

```
expr
```

or

```
lower-bound-expr TO upper-bound-expr
```

Individual values and value ranges can be intermixed freely.

It works this way: First, ***expr***$_0$ is evaluated. Each ***case-expr*** in each **CASE** statement is examined in turn. If the ***case-expr*** is an individual value, then if ***expr***$_0$ is exactly equal to that value, then the immediately following ***statements*** will be performed; or, if ***expr***$_0$ is equal to or greater than ***lower-bound-expr*** and less than or equal to ***upper-bound-expr***, then the immediately following ***statements*** will be performed.

If none of the ordinary **CASE** statements match ***expr***$_0$, but there is a **CASE ELSE** statement, the immediately following ***statements*** will be performed.

If and when a matching **CASE** is encountered, and the immediately following ***statements*** are peformed, all further statements, including **CASE** statements, will be ignored until the matching **END SELECT** statement.

The **SELECT** statement may be thought of as an "express" version of a sequence of **IF** ... **THEN** ... **ELSE IF** (...) **END IF** statements where (a) the initial **IF** comparison and all the **ELSE IF** comparisons all involve the same variable or expression, and (b) the comparison is always one of equality. Rather than repeating that variable or expression and the equality operator, in the **SELECT** statement the expression is specified only once, and comparison of equality is implied.

Examples:

The following examines a simple variable, and compares it to a range, and to members of a list, and does something different in each case; if neither case applies the **CASE ELSE** statement does something else entirely.

```
SELECT X
  CASE 1 to 3                    // Use a range.
      PRINT "would go at top"
  CASE 21, 22, 23            // Use a list.
      PRINT "would go at bottom"
  CASE ELSE                      // Catch all other cases.
      PRINT "would go in main area"
END SELECT
```

One can readily see that if X is outside the expected range (1 through 23, inclusive), this will behave badly. A better rendering would be:

```
SELECT X
    CASE 1 to 3
    PRINT "would go at top"
    CASE 4 TO 20
    PRINT "would go in main area"
    CASE 21 TO 23
    PRINT "would go at bottom"
    CASE ELSE
    PRINT "invalid value"
END SELECT
```

In the following example, there is no **CASE ELSE** statement. Because of this, if the variable does not match one of the six specified ranges, *nothing* happens.

```
SELECT X
  CASE 01 TO 03
      PRINT X;" is in ";"first three years";" of first decade"
      CASE 11 TO 13
      PRINT X;" is in ";"first three years";" of second decade"
      CASE 21 TO 23
      PRINT X;" is in ";"first three years";" of third decade"
      CASE 31 TO 33
      PRINT X;" is in ";"first three years";" of fourth decade"
      CASE 41 TO 43
      PRINT X;" is in ";"first three years";" of fifth decade"
      CASE 51 TO 53
      PRINT X;" is in ";"first three years";" of sixth decade"
END SELECT
```

3.3.5 FOR

Syntax:

FOR *variable*$_1$ = *expr*$_1$ TO *expr*$_2$ [**STEP *expr*$_3$**]
statements
NEXT *variable*$_1$

This statement is the way to do one or more statements over and over again, a certain number of times, each time setting the value of some variable to a new value.

First, the variable is set of the value of the first expression. Then the statements in the middle are executed. The **NEXT** statement indicates that the variable (note that this is the same variable that is part of the **FOR** statement) should be set to the next value; if the new value of the variable is greater than the second expression, the statements in the middle are skipped, and the next statement to be executed will be the one immediately following the **NEXT** statement.

By default – i.e.if the STEP clause is omitted – the next value is always one (integer 1) greater than the previous value.

The variable may be used in the statements between the FOR and NEXT statements, or not; sometimes you only need it to control *how many times* a thing is done, not use it for anything else.

Examples:

Print out the numbers from 1 to 10:

```
FOR j = 1 TO 10
  PRINT j
  NEXT j
```

Print out every 3rd number from 1 to 20 (1, 4, 7, 10, 13, 16, and 19):

```
FOR j = 1 TO 20 STEP 3
  PRINT j
  NEXT j
```

3.3.6 DO

Syntax:

```
DO { UNTIL | WHILE } expr
statements
LOOP
```

or

```
DO
statements
LOOP { UNTIL | WHILE } expr
```

This statement is the way to do one or more statements over and over again, based on very general criteria.

In either case in which the test is (or appears) syntactically *before* the controlled statements (that is, **DO UNTIL *expr*** or **DO WHILE *expr***), the test is performed prior to executing the statements.

In either case in which the test is (or appears) syntactically *after* the controlled statements (that is, **LOOP UNTIL *expr*** or **LOOP WHILE *expr***) the test is performed after executing the statements and therefore the loop executes at least once no matter what the value of the expression is.

The difference between **WHILE** and **UNTIL** is that **WHILE** performs the controlled statements as long as the value of the test expression remains true, whereas **UNTIL** performs the controlled statements as long as the value of the test expression remains false.

IMPORTANT: Unlike the **FOR** statement, the **DO** statement in all its forms can very easily become an "infinite", i.e. never-ending, loop! Specifically, if no statement(s) inside the loop alter any of the variables that make up the expression in the **DO** or **LOOP** statement, then the expression will never be altered, and can never become true (for **UNTIL**) or false (for **WHILE**). Even changing one or more variables that make up the expression doesn't guarantee that the expression will change from false to true or vice versa, so considerable care is required.

Examples:

Get 128 bytes of data from somewhere (using a user-defined function):

```
byteCount = 0
DO UNTIL byteCount = 128
  CALL loadByte()
  byteCount = byteCount + 1
  LOOP
```

Get bytes of data from somewhere (using a user-defined function) until an EOF byte is encountered. As each byte comes in, store it in a buffer, and keep a count. Don't store the EOF in the buffer or include it in the count:

```
DEF EOF = 0x0F
i = 1
DO UNTIL byte = EOF
  byte = getByte()
  if byte != EOF THEN
     buffer[i] = byte
     i = i + 1
     END IF
  LOOP
byteCount = i - 1
```

4 Built-In Functions

Chameleon Basic has a number of built-in functions. Here is a list, by category:

- String functions

  LEFT$ (***str*, *n*** **)**
  RIGHT$ (***str*, *n*** **)**
  MID$ (***str*, *start*, *n*** **)**
  LEN (***str*** **)**
  VAL (***str*** **)**
  STR$ (***n*** **)**
  ASC (***str*** **)**
  CHR$ (***n*** **)**

- Other Functions

  ABS (***numeric-expr*** **)**
  RND (***limit*** **)**

4.1 String Functions

concatenation
LEFT$
RIGHT$
MID$
LEN
VAL
STR$
ASC
CHR$

Note that the names of functions that return a string value end with a dollar sign.

4.1.1 concatenation

This is not really a function, but rather an operator. It concatenates the two strings that are on each side of it into one longer string.

Syntax:[8]

str_1 + str_2

This concatenates ***str1*** and ***str_2***, and has the value of the resulting string.

Examples

```
This expression:              has this value:
"American" + "Bandstand"      "AmericanBandstand"
A$="American"                 // Just assigning one variable.
B$="Bandstand"                // Just assigning another.
A$ + " " + B$                 "American Bandstand"
```

8 The **+** operator is also used as the numeric addition operator. Note that it cannot be used to add or concatenate a string and a number. It can only be used to add two numeric expressions or to concatenate two strings.

4.1.2 LEFT$

This function returns the leftmost ***n*** characters of the string ***str***. If ***n*** is greater than the length of the string, the function returns the entire string.

Syntax:

`LEFT$ ( str, n )`

Examples

This expression:	*has this value:*
`LEFT$("American", 4)`	`"Amer"`
`LEFT$("Bandstand", 4)`	`"Band"`
`LEFT$("Zeddicus", 4)`	`"Zedd"`

4.1.3 RIGHT$

This function returns the rightmost ***n*** characters of the string ***str***. If ***n*** is greater than the length of the string, the function returns the entire string.

Syntax:

`RIGHT$ ( str, n )`

Examples

This expression:	*has this value:*
`RIGHT$("American", 4)`	`"ican"`
`RIGHT$("Bandstand", 4)`	`"tand"`
`RIGHT$("Zeddicus", 4)`	`"icus"`

4.1.4 MID$

This function returns the ***n*** characters of the string ***str*** beginning at ***start***. If there are not ***n*** characters beginning at ***start***, the function returns the remainder of the string beginning at ***start***.

Syntax:

MID$ (*str*, *start*, *n*)

Examples

This expression:	*has this value:*
`MID$("abcdefghij", 3, 4)`	`"cdef"`
`MID$("American", 3, 4)`	`"eric"`
`MID$("Bandstand", 3, 4)`	`"ndst"`
`MID$("Zeddicus", 3, 4)`	`"ddic"`

4.1.5 LEN

This function returns the length of the string ***str***.

Syntax:

LEN (*str*)

Examples

This expression:	*has this value:*
`LEN("abcdefghij")`	10
`LEN("American")`	8
`LEN("Bandstand")`	9
`LEN("Zeddicus")`	8

4.1.6 VAL

This function returns the numeric value of the string ***str***; that is, it converts the string ***str*** to an integer.

Note: The string str must consist solely of digits, optionally preceded by a sign (+ or -), and optionally preceded and/or followed by blanks.

Syntax:

VAL (*str*)

Examples

This expression:	*has this value:*
`VAL("0000")`	0
`VAL("-0")`	0
`VAL("256")`	256
`VAL("255")`	255
`VAL("257")`	257
`VAL("32767")`	32767
`VAL("32766")`	32766
`VAL("32768")`	-32768

4.1.7 STR$

This function returns a string whose digits are the value of the integer ***n***.

Syntax:

STR$ (*n*)

Examples

This expression:	*has this value:*
`STR$(0)`	"0"
`STR$(256)`	"256"
`STR$(255)`	"255"
`STR$(257)`	"257"
`STR$(32767)`	"32767"
`STR$(32766)`	"32766"
`STR$(-32768)`	"-32768"

4.1.8 ASC

This function returns the ASCII code of the first character of the string ***str*** as an integer.

Syntax:

```
ASC ( str )
```

Examples

This expression:	has this value:
`ASC("American")`	65
`ASC("Bandstand")`	66
`ASC("Zeddicus")`	90
`ASC("    ")`	32

4.1.9 CHR$

This function returns a single-character string consisting of just the character whose ASCII code is ***n***.

Syntax:

```
CHR$ ( n )
```

Examples

This expression:	has this value:
`CHR$(64)`	`"@"`
`CHR$(65)`	`"A"`
`CHR$(66)`	`"B"`
`CHR$(67)`	`"C"`
`CHR$(68)`	`"D"`
`CHR$(69)`	`"E"`
`CHR$(70)`	`"F"`
`CHR$(71)`	`"G"`

4.2 Other Functions

ABS
RND

4.2.1 ABS

This function returns the absolute value of the specified (numeric) expression.

Syntax:

ABS (***expr*** **)**

Examples:

This expression:	has this value:
ABS (0)	0
ABS (1)	1
ABS (-1)	1
ABS (13)	13
ABS (-13)	13

4.2.2 RND

This function returns a pseudo-random number in the range **0** (zero) and **(*limit* – 1)**, inclusive. (In other words, it is equivalent to the C expression "rand() % n".)

Note: Because this generates a *pseudo*-random number rather than a *truly* random number, the actual values returned and the order in which they are returned will be the same each time the program is run.

Syntax:

RND (***limit*** **)**

Examples:

This expression:	has this value:
RND (9)	5
RND (9)	8
RND (9)	7
RND (9)	4
RND (9)	8
RND (99)	82
RND (99)	93
RND (99)	54
RND (99)	34
RND (99)	11

5 Language Summary

This section summarizes the entire syntax of Chameleon Basic, using a format very similar to one known as Backus-Naur Form, or BNF. In each definition, or “production”, the first term is the one being defined, and it is shown in normal typeface.

The actual syntax is shown in **bold face**.

By contrast, the meta-syntax – those characters indicating denoting which pieces of actual syntax are optional or alternatives – are shown in normal case.

Keywords are shown in **ALL-UPPERCASE**, although (as noted above) this is not a requirement of the language; it's just used here to help distinguish keywords from things that are not keywords.

Terms that require further definition, and are defined below where they are used, are shown in ***italics***.

As with BNF, brackets ('[' and ']') enclose optional pieces of syntax – you can include them, or leave them out, either at your whim or as appropriate to the situation. Braces ('{' and '}') enclose sets of alternatives, each alternative separated from its neighbor(s) by a vertical bar ('|'). A trio of dots or periods ('...') is used to indicate that the previous piece of syntax may be repeated any number of times.

5.1 Labels

Any statement may be preceeded by an identifier followed by a colon. This is called a *label* and can be the target of a **GOTO** statement.

5.2 Statements

```
statements ::=
                statement
    |           statements
```

Note: This definition is somewhat informal. It means that the word “statements” (plural) as used in the syntax descriptions above mean either a single statement or more than one statement, each on a line by itself.

5.3 Statement

```
statement ::=
        |    REM comment text to end of line
        |    OPTION TARGET = { "tile" | "bitmap" }
        |    DEF name = value
        |    DEF name ( [ arg [ , arg ] ... ] )
        |    END DEF
        |    SUB name ( [ arg [ , arg ] ... ] )
        |    END SUB
        |    DIM variable-defs
        |    [ LET ] l-value = expr
        |    IF expr THEN statement
        |    IF expr THEN
        |    ELSE IF expr THEN
        |    ELSE
        |    END IF
        |    SELECT
        |    CASE
        |    CASE ELSE
        |    END SELECT
        |    FOR var = expr TO expr [ STEP expr ]
        |    NEXT var
        |    DO
        |    DO WHILE expr
        |    DO UNTIL expr
        |    LOOP
        |    LOOP WHILE expr
        |    LOOP UNTIL expr
        |    GOTO label
        |    [ CALL ] name [ ( [ arg [ , arg ] ... ] ) ]
        |    PRINT
        |    STOP
        |    END
```

5.4 variable-defs

```
variable-defs ::=
                variable-def
          |     variable-defs , variable-def
```

5.5 variable-def

```
variable-def ::=
                basic-variable-def [ = initializer ]
```

5.6 basic-variable-def

```
basic-variable-def ::=
                variable [ AS type ]
          |     variable ( size [ , size ] ) [ AS type ]
```

5.7 type

```
type ::=
                BYTE
          |     FLOAT (not yet implemented)
          |     INTEGER
          |     STRING[9]
```

5.8 initializer

```
initializer ::=
                init-expr
          |     { init-expr [ , init-expr ] ... }
```

5.9 init-expr

init-expr ::=

*an expression composed of integer constants and possible also simple arithmetic operators (+, -, *, /)*

9 As of this version (1.4) strings are not implemented except as string constants.

5.10 expr

In what follows, it may not always be clear that the punctuation marks that either are between one *expr* and another, or precede the *expr*, or surround the *expr*, are in bold face. They are, just like the keywords OR, AND, MOD, and so forth. As such they are required. Likewise it may not be clear that "0x" is in bold face. It is, and is a required part of hexadecimal constant.

```
expr ::=
              expr OR expr
        |     expr AND expr
        |     expr ^ expr
        |     expr | expr
        |     expr & expr
        |     expr = expr
        |     expr <> expr
        |     expr < expr
        |     expr <= expr
        |     expr >= expr
        |     expr > expr
        |     expr << expr
        |     expr >> expr
        |     expr + expr
        |     expr - expr
        |     expr * expr
        |     expr / expr
        |     expr MOD expr
        |     - expr
        |     NOT expr
        |     ~ expr
        |     ( expr )
        |     decimal-constant
        |     0xhex-constant
        |     string
        |     variable
        |     array-reference
        |     function-call
```

5.11 l-value

```
l-value ::=
              array-reference
        |     variable
```

5.12 array-reference

```
array-reference ::=
              variable ( index [ , index ] )
```

5.13 variable

```
variable ::=
              name
```

5.14 function-call

```
function-call ::=
              name [ ( [ arg [ , arg ] ... ] ) ]
```

Note: The argument list, including the parentheses, may be omitted IFF the function does not require any arguments.

5.15 built-in Propeller SPI functions

```
PROP ( fcn )
PROP_8 ( fcn, 8-bit-arg )
PROP_8_8 ( fcn, 8-bit-arg, 8-bit-arg )
PROP_16 ( fcn, 16-bit-arg )
PROP_RESULT
```

5.16 built-in tile graphics functions

```
GFX_DEFINE_TILE_MAP ( map-number, offset, width, height )
GFX_DISPLAY_TILE_MAP ( map-number, x, y )
GFX_SET_TERM_TILE_MAP ( map-number )
GFX_LOAD_TILE ( tile-number, tile-array )
GFX_LOAD_PALETTE ( palette-number palette-array )
GFX_WRITE_TILE ( tile-map-number, x, y, palette-and-tile-number )
GFX_READ_TILE ( tile-map-number, x, y, palette-and-tile-number )
```

5.17 other built-in functions

```
RND ( limit )
```

5.18 name

```
name ::=
                letter
        |       letter alphanums
```

5.19 decimal-constant

```
decimal-constant ::=
                [ sign ] digit-string
```

Note: The value of a decimal-constant must be in the range -32768 through 32767, inclusive. Spaces are not allowed within a decimal-constant.

5.20 digit-string

```
digit-string ::=
                digit
        |       digit digit-string
```

Note: Spaces are not allowed within a digit-string.

5.21 hex-constant

```
hex-constant ::=
                hex-digit
        |       hex-digit hex-constant
```

Note: The value of a hex-constant must be in the range 0x0000 through 0xFFFF, inclusive. Spaces are not allowed within a hex-constant.

5.22 string-constant

```
string-constant ::=
                " printable-characters "
```

Note: There is no specific limit to the length of a strong constant, only the practical limit of the available memory. The doublequotes, one at each end of the string constant, are required.

5.23 printable-characters

```
printable-characters ::=
            printable-characters
      |     printable-character printable-characters
```

5.24 printable-character

```
printable-character ::=
            letter
      |     digit
      |     punctuation-mark
      |     blank
```

5.25 alphanums

```
alphanums ::=
            alphanum
      |     alphanum alphanums
```

5.26 alphanum

```
alphanum ::= letter | digit
```

The following define the which specific characters make up the syntactic items above.

5.27 letter

```
letter ::=
                A | B | C | D | E | F | G | H | I | J | K | L | M
        |       N | O | P | Q | R | S | T | U | V | W | X | Y | z
        |       a | b | c | d | e | f | g | h | i | j | k | l | m
        |       n | o | p | q | r | s | t | u | v | w | x | y | z
```

5.28 punctuation-mark

```
punctuation-mark ::=
                . | , | : | ; | ! | ? | / | \ | '
        |       ` | ~ | @ | # | $ | % | ^ | & | *
        |       _ | + | - | = | ( | ) | { | } | [ | ] |
```

5.29 hex-digit

```
hex-digit ::=
                A | B | C | D | E | F | a | b | c | d | e | f
        |       digit
```

5.30 digit

```
digit ::=       0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```